Going to Extremes

James W. Cooper

Sometimes you write some code and pass it on to someone else, who writes some more code, who passes that on and so forth. At the end of the chain someone takes the result of this digital version of "whispering down the lane" and tries to make all of it work. Sometimes it does, and frequently it doesn't. Maybe they'll come back to you for help and maybe they'll try to fix it themselves. But in the meantime, you've continued to work on that algorithm and have a better version now. Can they use that one? Maybe. Maybe not.

In worst-case scenarios, whole projects get structured this way (and yes I've seen some of them) and while they might not bog down immediately, the long-term prospects for success are significantly reduced.

This kind of experience reminds me of Kent Beck's description of the discipline of Extreme Programming (XP), which despite its name, is definitely a set of really good ideas for code writing. When you first hear the name "Extreme programming," you tend to think of kneepads and big concave concrete bowls filled with people on in-line skates. But while Extreme Programming is about being wild, it really attempts to codify the behavior of "programmers in the wild."

The XP discipline grew out of Beck's experiences in trying right a seriously off-course project, the Chrysler Comprehensive Compensation system. They transformed the seemingly unredeemable project into one that was delivered ahead of schedule using a combination of conventional OO and XP techniques.

There are two major precepts that Beck presents as the most significant parts of Extreme Programming: test cases and pair programming. First is the somewhat heretical idea that you must have tests for the program before you even write it, and that writing the tests first and the program second is a sensible pursuit.

Testing Your System

There are two kinds of tests to consider, unit tests and functional tests. A unit test may sound like a mind-numbing concept, but it just means that every class should be able to test itself. We don't usually write code that way on a day-to-day basis, but if you are building a system to last more than a few days, it's a good idea. Building a test case into each class may waste time now, but save time next month.

A functional test is a little harder to picture. In that case it amounts t some sort of script that puts a program through its paces. In Java, that probably means a self-execution module that calls various program functions and compares the results. In a GUI environment, it might even be a system that executes that GUI untouched by human hands. This sounds like a lot of work for a simple program, but starts to pay off in more complicated cases. You can start to see to trade-offs, though. If you are writing payroll modules, Beck's favorite example, you can easily compare some canned data with some canned results. If you are building an interactive interface and adding new UI features this can be a real p-i-t-a.

Getting to Know Your Buddies

More useful, but equally foreign to many of us, is the idea of pair programming. The scenario is simple. Two programmers sit together, one driving the mouse and keyboard and the other sitting alongside and watching and conversing. The one doing the "driving" concerns himself with the details of coding, and the one alongside thinks about larger structural issues. He can also point out bugs before they get committed. At any time the two can change places by just sliding the keyboard from one to the other. Note, then, that this is not one supervising the other or looking over the other's shoulder (this would be really annoying) but an even one-on-one exchange of ideas.

Further, the pairs of programmers can change over time. One pair may form in the morning, and those two may be part of two other pairs in the afternoon. Obviously, this takes place best in an environment where there are a number of team members working on parts of a larger project.

Finally, as you might expect, in this sort of environment, you have to have some agreed-upon coding standards. While programmers might rather die than put their curly braces somewhere else, Beck notes that being part of a winning team makes the imposition of such standards pretty easy to live with. And of course, team programming is pretty much impossible without it.

But that's *my* code!

Of course, the objection you might immediately raise is: who gets to change what code? Code ownership diminishes in this approach, and anyone can in theory change anyone else's code (with appropriate source code control in place). Of course some people will be more familiar with some parts of the system than with others, but any pair can look at a design and change it if it makes the system more versatile or reliable. You might think this means that some dummy could change your carefully crafted and beautifully written system into a pile of shredded carrots, but when pair programming prevails there are some checks and balances on these changes. Two people, the code changer and the architectural thinker have to agree on any such changes, and they have to test the result. So they might (god forbid!) improve your code. Wouldn't that be nice?

But Does it Work?

Theories are great, but results are better. If you remember my March, 2001 column on Tichy, Prechelt and Unger's experiments on whether Design Patterns work, they did an experiment where they had an experienced and less-experienced programmer collaborate on a project before and after learning the lingo of Design Patterns. One of the outcomes of this experiment was that the less experienced programmer became a contributor more rapidly once they learned this common vocabulary. In other words, the act of conversing made them better programmers.

In a more significant experiment, Alistair Cockburn and Laurie Williams performed some controlled studies of pair programming. They first note a report from the field: from a programmer whose organization began to undertake pair programming. At first they reported that programmers continued to work individually and then meet to review their code. But after 3-4 months, the culture began to change, and "pairs began honest-to-

goodness pair programming." They note that the reasons for this were clear: it worked better. Rather than coding and then meeting afterwards, they combined the two functions and wrote better code in the process.

In Cockburn and Williams' experiments, they took students from a programming course at the University of Utah, and had half of them work in pairs and half work individually. They found that while the code produced by the teams took about 15% longer than that produced by the individuals, *it contained 15% fewer errors!*

This answers the most frequent management objection. Pair programming doesn't cut your productivity in half: it improves it! Defects in code cost much more to fix further down the line, and any software developer can tell you, and producing 15% fewer errors up front is a lot more than 15% cheaper in the long run.

Even more interesting is the fact that the programs constructed by pairs in every case contained significantly (15-25%) fewer lines of code. If this is an indication of probable code quality, it is a pretty persuasive one.

But I know *I* wouldn't like it!

Yeah. I wouldn't either. But experiments showed that pair programming teams reported that they enjoyed programming that way much more than programming alone after they got over the initial adjustment hump. In fact, one pair noted that it was nice to celebrate with somebody when something works.

Other Facets of Extreme Programming

To go along with the idea of pair programming and frequent changes of pairs, Beck recommends that the programmers mainly work in workrooms where they can easily help each other ("anyone know how do this…?"). They also recommend small, incremental releases, each with its own set of tests, and that the end customers and their requirements become part of the development project. Their "requirements stories" can lead to better testing and understanding of the necessary design.

In addition, Beck recommends refactoring the code whenever necessary. I discussed refactoring in my August column in some detail, but it really just means revising the classes to make them more efficient and versatile whenever necessary. In the XP discipline, you do this much more frequently than you might in a system where code architecture is frozen at the outset.

Finally, Beck strongly emphasizes the 40-hour week as a standard. You can work overtime once in a great while, but if it becomes standard practice, you and your projects are in a lot a trouble. Teams need to come to their code refreshed and ready to dig in.

The complete list of the 12 guidelines for XP is:

1. Planning with customers
2. Functional testing and unit testing
3. Refactoring
4. Simple design

5. Simple code metaphors

6. Collective code ownership

7. Coding standards

8. Continuous integration of changes

9. Customer on site.

10. Open workspace

11. Forty Hour week

12. Pair programming

But, will it work for me?

I don't know. In our environment, we tend to have one-person projects, not big team efforts, but then, if I think about the number of times I have been in someone else's office (and vice-versa) discussing why their code (or mine) doesn't work, I realize that even in the prima-donna environment, pair programming makes sense.

Where the whole XP idea may break down is in the go-go job market (we used to have one anyway), where people come and go fairly frequently. In that case the concept of a winning project or team is harder to maintain and bringing new people up to speed becomes an additional cost to the whole team.

Some places where XP probably won't work is when the programming team is very large (say more than 20 or so), or when they are not all located at the same physical facility. Beck also notes that business or management culture may make XP fail. If they insist on a complete specification before starting, XP can never take hold, and if management insists that the team not steer the projects as it evolves, this can be a problem. Finally, you really need a work environment where teams can form. Ideally, a series of offices surrounding an open work area engenders good XP. And, if management insists on long hours, this will make teams too tired to work effectively. But, for the most part, I think it's worth trying.

References

1. Kent Beck, *Extreme Programming Explained: embrace change,* Addison-Wesley, Reading, MA, 2000.

2. Alistair Cockburn and Laurie Williams, "The Costs and Benefits of Pair Programming," http://members.aol.com.humansandt/papers/pairprogrammingcostbene/pairprogrammingcostbene.htm

3. "Extreme Programming," http://ootips.org/xp.html